NPS52-85-015

NAVAL POSTGRADUATE SCHOOL

Monterey, California

EXPERIENCE WITH Ω IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT PART II

Bruce J. MacLennan

December 1985

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by Contract from the Office of Naval Research.

This report was prepared by:

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-85-015	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: EXPERIENCE WITH Ω IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT PART II	5. TYPE OF REPORT & PERIOD COVERED
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Bruce J. MacLennan	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93943-5100	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61152N; RR000-01-NP, N0001485WR41005
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217	12. REPORT DATE: December 1985
	13. NUMBER OF PAGES: 37
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This is the second report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report extends the interpreter, unparser, syntax directed editor, command interpreter and debugger to accomodate block-structured identifier declaration and reference.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

EXPERIENCE WITH Ω

IMPLEMENTATION OF A

PROTOTYPE PROGRAMMING ENVIRONMENT

PART II

Bruce J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract:

This is the second report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report extends the interpreter, unparser, syntax directed editor, command interpreter and debugger to accomodate block-structured identifier declaration and reference.

1. Introduction

Our goal in this series of reports* is to explore in the context of a very simple language the use of the Ω programming notation [MacLennan83, MacLennan85] to implement some of the tools that constitute a programming environment.

Our goal in this report is to extend the language defined in [MacLennan85b] to incorporate block structured name definitions. For example, we want to allow programs such as these:

```
[let X = (3+5)
 [let Y = (6÷2)
  (X+(Y-1)) ] ]
```

We do this in stages, first making the necessary changes to the abstract syntax, second modifying the unparser and syntax-directed editor, third extending the evaluator, and finally modifying the debugger.

* Support for this research was provided by the Office of Naval Research under contracts N00014-85-WR-24057 and N00014-86-WR-24092.

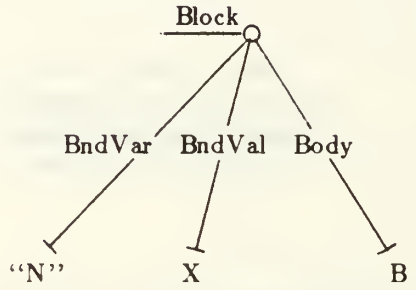
2. Abstract Structure

There are two new constructs to be added to the language: **let** blocks for binding names to values, and identifier references for making use of the values bound to names. A **let** block such as this:

[**let** N = X

B]

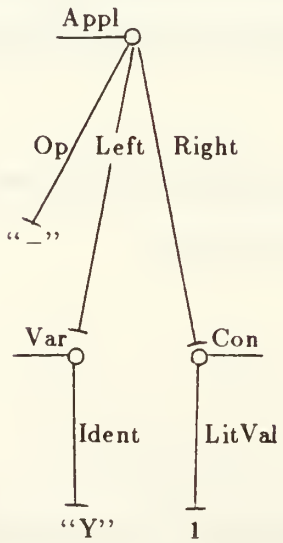
will be represented by the structure



Identifier references are illustrated by the expression:

(Y−1)

which is represented by the abstract structure:



These abstract structures are easily accommodated in the relational framework. The *static relations* required are:

- Block (E)

  E is a block

- BndVar (N, E)

  N is bound variable of E

- BndVal (X, E)

  X is bound value of E

- Body (B, E)

  B is body of E

- Var (E)

  E is a variable

- Ident (N, E)

  N is identifier of E

The domains of these relations are:

- expr = Con $\cup$ Appl $\cup$ Block $\cup$ Var $\cup$ Undef
- Degree (Block, 1)
- Function (BndVar, Block, string)
- Function (BndVal, Block, expr)
- Function (Body, Block, expr)
- Degree (Var, 1)
- Function (Ident, Var, string)

3. Unparser

3.1 Unparsing Variables

The unparser rules must be augmented to accommodate the new constructs. Since Var nodes are leaves in the expression tree, they can be handled by a single rule:

*Unparse (E), Var (E), Ident (N, E)

$\Rightarrow$ Image (N, E).

That is, if we are unparsing a variable, and a string is the identifier associated with that variable, then that string becomes the image of that variable.

3.2 Unparsing Blocks

Since Block nodes are not leaves, two additional rules are required, an *analysis* rule that passes the Unparse attribute down the tree, and a *synthesis* rule that forms the composite image from the images of the components. The analysis rule is:

*Unparse (E), Block (E), BndVal (X, E), Body (B, E)

$\Rightarrow$ Unparse (X), Unparse (B).

That is, when we are unparsing a block, then we unparse the bound value and the body of that block. We do not need to unparse the bound variable, since that is just a string, i.e., its own image.

The synthesis rule is:

Block (E), BndVar (N, E), BndVal (X, E), Body (B, E), *Image (U, X), *Image (V, B)

$\Rightarrow$ Image (

TabIn ^NL

^"[**let** " ^N ^" = " ^U

^TabIn ^NL ^V ^"]"

^TabOut ^TabOut,

E).

That is, when images have arrived for the bound value and body of a block, then we use these to con-

struct the image for the block. Notice that we have assumed that the variables 'NL', 'TabIn' and 'TabOut' are defined. These are strings that represent commands to a low-level display formatter, which is also assumed to break between tokens lines that are too long. The functions of these command strings are:

- NL: Go to new line and start to display text at current left margin.
- TabIn: Move left margin in (to the right) by some fixed increment (say, two spaces).
- TabOut: Move left margin out (to the left) by the same fixed increment used by TabIn.

TabIn and TabOut are assumed to take effect on the next newline; they do not cause a newline on their own.

4. Syntax-Directed Editor

In this section we consider the changes to the syntax-directed editor necessary to accommodate Block and Variable nodes. First, the **let** command for creating Block nodes is handled by the rule:

*Command (**let**), *Argument (N), *CurrentNode (E), *Undef (E), *Avail (X,B)

$\Rightarrow$ Block (E), BndVar (N, E), BndVal (X, E), Body (B, E),

Undef (X), Undef (B), CurrentNode (X).

The rule for the **var** command is similar:

*Command (**var**), *Argument (N), CurrentNode (E), *Undef (E)

$\Rightarrow$ Var (E), Ident (N, E).

It would also be desirable to have rules that detect the attempt to make an already-defined node into a Block or Var. Furthermore, rules are required for deleting Block and Var nodes.

It is also necessary to update the local navigation commands (**in**, **out**, **next** and **prev**) to recognize Blocks. This requires at least one extra rule for each of these commands. For example, for **next** we need:

*Command (**next**), *CurrentNode (X), BndVal (X, E), Body (B, E)

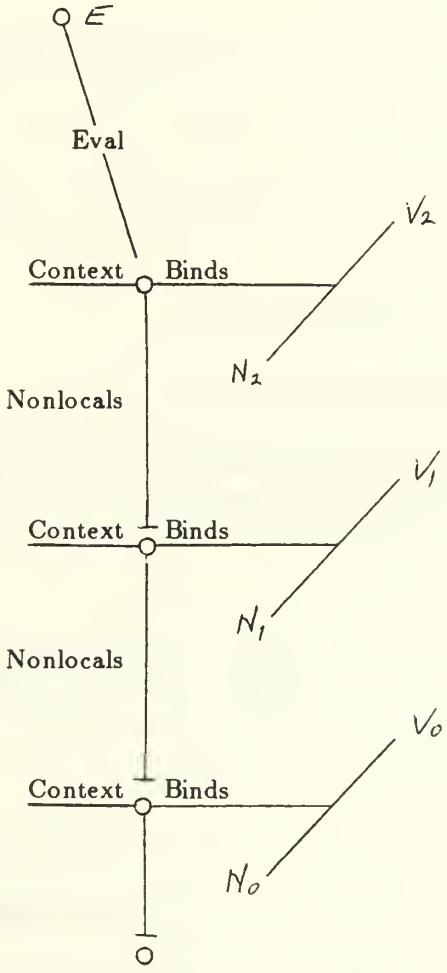
$\Rightarrow$ CurrentNode (B), Command (**show**).

The other commands are similar.

5. Dynamic Structure

5.1 Data Structures

Clearly changes will have to be made to the interpreter to accommodate the new program structures. It will now be necessary to keep track of the *environment* in which an expression is to be evaluated. Here the environment is a set of nested *contexts,* each of which *binds* a name to its value. Thus the environment can be visualized as a chain, beginning with the environment of evaluations and terminating with the outermost (empty) context:



This runtime structure can be represented by the following dynamic relations:

- Context(C)

  C is a context

- Binds(C, N, V)

  C binds N to V

- Nonlocal(C,D)

  C is nonlocal context of D

The domains of these relations are:

Degree (Context,1)

- Degree (Binds,3)
- Domain (1, Binds, Context)
- Domain (2, Binds, string)
- Domain (3, Binds, integer)
- Indexed (Binds, 1)
- Function (Nonlocal, Context, Context)

5.2 Modifications to Existing Interpreter Rules

Next we consider the required modifications to the interpreter. In particular, we must modify the interpreter to reflect context in which evaluation is being done. Thus we add a 'C' parameter to every existing use of Eval relation:

$$\text{'Eval(' - ')'} \quad \Rightarrow \quad \text{'Eval(' - ',}C\text{)'}.$$

(Note that (E,C) arguments to Eval are analogous to an IP-EP pair.) We make analogous changes to Value and Check:

$$\text{'Value (' - ',' - ')'} \quad \Rightarrow \quad \text{'Value (' - ',' - ',}C\text{)'}.$$

$$\text{'Check (' - ',' - ')'} \quad \Rightarrow \quad \text{'Check (' - ',' - ',}C\text{)'}.$$

Since the interpreter may detect an error in evaluation, we must also consider the proper handling of the environment when an error occurs. Hence, in addition to recording (in CurrentNode) the node whose evaluation caused the error, we will also record (in CurrentContext) the context in which that node was being evaluated. So we alter error handling to save the current node *and* the current context:

*Check(W,E,C), Explanation(S,W), *CurrentNode(-), *CurrentContext(-) **if** errorcode[W]

$\Rightarrow$ Display(S), CurrentNode(E), CurrentContext(C).

Similarly, the **evaluate** command uses this saved context:

*Command(**evaluate**), CurrentNode(E), CurrentContext(C)

$\Rightarrow$ Eval(E, C).

5.3 Rules for Block Evaluation

There are three steps in the evaluation of a block:

1. Evaluate the bound value expression.
2. Bind the local name to the bound value and evaluate the block body in the resulting context.
3. Return the value of the block body to the parent expression.

We consider the rules for each of these steps. To evaluate the bound value we have:

*Eval(E,C), Block(E), BndVal(X,E)

$\Rightarrow$ Eval(X, C).

That is, when the Eval message arrives at the block, we send it on to the bound value expression. Notice that the bound value expression is evaluated in the same environment as the block (thus recursive definitions are not permitted).

Next we bind the local name to the bound value and evaluate the body in the resulting context. This is accomplished by waiting for a value to arrive and the node for the bound value expression, and using it to create a new context:

Block(E), BndVar(N,E), BndVal(X,E), Body(B,E), *Value(V,X,C), *Avail(D)

$\Rightarrow$ Context(D), Binds(D,N,V), Nonlocals(C,D), Eval(B,D).

The last rule waits for a value to be attached to the block body, and returns the value to the surrounding expression:

Block(E), Body(B,E), *Value(V,B,D), NonLocals(C,D)

$\Rightarrow$ Value(V,E,C).

5.4 Name Lookup

Next we consider the rules for name lookup. To control the lookup process we need an additional dynamic relation:

- Looking(N, C, E, D)

  Looking for N in C, to be value of E in D

- Degree (Looking,2)
- Domain (1, Looking, string)
- Domain (2, Looking, Context)
- Domain (3, Looking, expr)
- Domain (4, Looking, Context)

The first rule initiates the search when an Eval message arrives at a Var node. The search begins with the current context:

*Eval(E, C), Var(E), Ident(N, E)

$\Rightarrow$ Looking(N, C, E, C)

Now there are several possible conditions that occur during a search. First, the binding may be found, in which case we must attach the bound value to the Var node:

*Looking(N, C, E, D), Binds(C, N, V)

$\Rightarrow$ Value(V, E, D).

Second, we might not have found the binding, but there might be more nonlocal contexts in the environment chain. Thus we continue looking:

else *Looking(N, C, E, D), Nonlocal(C', C)

$\Rightarrow$ Looking(N, C', E, D).

Finally, if a binding for the name has not been found, and we are at the end of the environment chain, then the variable must be unbound, so we signal an error:

else *Looking(N, C, E, D), *CurrentNode($-$), *CurrentContext()

$\Rightarrow$ CurrentNode(E), CurrentContext(D), Display("Unbound: " ^N).

Thus we are naturally led to the next topic, the debugger.

6. Debugging Commands

6.1 Desired Features

Debugging requires commands to interrogate and update the "contextual database." For example:

- **out_context**:

  move to next outer context

- **in_context**:

  move to next inner context

  (this may not be single-valued, i.e., there may be several more inner contexts; a real debugger would have to handle this)

- **context**:

  display local context

- **alter** *v*:

  alter local binding

We give an example illustrating these commands. For the sake of the example we suppose that the evaluation is started and then interrupted (by the **attention** key) within the innermost context. We begin by showing the program:

```
show
  [let X = (3+ 5)
    [let Y = (6÷2)
      (X+ (Y−1)) ] ]
```

Next we begin interpretation, and interrupt it:

```
evaluate
attention
  interrupted
```

We can now look at the current context (saved in CurrentContext upon interruption):

context

Y = 3

We can move to the surrounding context:

out_context

X = 8

We can alter the bound value in this context:

alter 7

X = 7

We can return back to the original context:

in_context

Y = 3

And, if we move too far out, we'll be told:

out_context

X = 7

out_context

at outermost level

6.2 Implementation of Debugging Commands

How can these commands be implemented? They're quite easy. First consider the **context** command. All that's necessary here is to extract the bound variable and it's value from the current context, and display them:

*Command(**context**), CurrentContext(C), Binds(C,N,V)

$\Rightarrow$ Display(N ^" = " ^string$\leftarrow$int[V])

else *Command(**context**)

$\Rightarrow$ Display("no bindings").

We have added a diagnostic rule for the case where there are no active contexts.

The **out_context** command is also simple: we move out on the environment chain and display its binding by issuing a **context** command:

*Command(**out_context**), *CurrentContext(D), Nonlocal(C,D)

$\Rightarrow$ CurrentContext(C), Command(**context**)

else *Command(**out_context**)

$\Rightarrow$ Display("at outermost level").

The **in_context** command is analogous.

All that remains is the **alter** command. In this case we expect a new value to be typed in, and then use this value to replace the previous bound value:

*Command(**alter**), *Argument(U), CurrentContext(C), *Binds(C,N,V)

$\Rightarrow$ Binds(C,N,U), Command(**context**)

else *Command(**alter**), *Argument($-$)

$\Rightarrow$ Display("no binding").

Again we've added a diagnostic rule.

7. References

[MacLennan83] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.

[MacLennan84] MacLennan, B. J., The Four Forms of Ω : Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.

[MacLennan85a] MacLennan, B. J., A Simple Software Environment Based on Objects and Relations, *Proc. of ACM SIGPLAN 85 Conf. on Language Issues in Prog. Environments,* June 25-28, 1985, and Naval Postgraduate School Computer Science Department Technical Report NPS52-85-005, April 1985.

[MacLennan85b] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part I, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-006, May 1985.

[McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter,* MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.

[Ufford85] Ufford, Robert P., *The Design and Analysis of a Stylized Natural Grammar for an Object-Oriented Language (Omega),* MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: Prototype Programming Environment

Predicate Notation for Ω

The following is a loadable input file for the prototype programming environment described in this report. It is accepted by the McArthur interpreter [McArthur84], which differs in a few details from the Ω described in the previous report (see [MacLennan84]). A transcript of a test execution of this environment is shown in Appendix B.

```
!            PI-2
!      Rules and associated definitions for
!      an arithmetic expression language
!      with variable declarations.


!      Relations

! Program Structure Relations

define {root, "Appl", newrel{}};
define {root, "Op", newrel{}};
define {root, "Left", newrel{}};
define {root, "Right", newrel{}};
define {root, "Con", newrel{}};
define {root, "Litval", newrel{}};
define {root, "Block", newrel{}};
define {root, "BndVar", newrel{}};
define {root, "BndVal", newrel{}};
define {root, "Body", newrel{}};
define {root, "Var", newrel{}};
define {root, "Ident", newrel{}};

! Evaluation Relations
```

```
define {root, "Eval", newrel{}};

define {root, "Check", newrel{}};

define {root, "Value", newrel{}};

define {root, "Meaning", newrel{}};

define {root, "Explanation", newrel{}};

define {root, "Context", newrel{}};

define {root, "Binds", newrel{}};

define {root, "Nonlocal", newrel{}};

define {root, "Looking", newrel{}}.

! Unparser Relations

define {root, "Unparse", newrel{}};

define {root, "Image", newrel{}};

define {root, "Template", newrel{}};


! Command Interpreter Relations

define {root, "Command", newrel{}};

define {root, "Argument", newrel{}};

define {root, "Root", newrel{}};

define {root, "Undef", newrel{}};

define {root, "CurrentNode", newrel{}};

define {root, "CurrentContext", newrel{}};

define {root, "EvalPending", newrel{}};

define {root, "ShowPending", newrel{}};

define {root, "CreateAppl", newrel{}};

define {root, "CreateRoot", newrel{}};

define {root, "CreateLet", newrel{}};
```

```
define {root, "CreateContext", newrel{}};

define {root, "Script", newrel{}};

define {root, "PendScript", newrel{}};

define {root, "Test", newrel{}}.


!      Functions

fn Id [x]: x;

fn Sum [x,y]: x + y;

fn Dif [x,y]: x - y;

fn Product [x,y]: x * y;

fn Quotient [x,y]:

   if y = 0 -> ["error", 1]

   else x / y;


fn IsErrorcode [w]:

   if ~IsList[w] | w = Nil -> Nil

   else first[w] = "error";


fn upSum [x,y]: "(" + x + " + " + y + ")";

fn upDif [x,y]: "(" + x + " - " + y + ")";

fn upProd [x,y]: "(" + x + " x " + y + ")";

fn upQuot [x,y]: "(" + x + " / " + y + ")".


! Constants

define {root, "NL", "

"};

define {root, "TabIn", ""};

define {root, "TabOut", ""};
```

```
!     Built-in Tables

Meaning (Sum, "+ ");

Meaning (Dif, "-");

Meaning (Product, "x");

Meaning (Quotient, "/");

Meaning (Id, "lit");

Template (upSum, "+ ");

Template (upDif, "-");

Template (upProd, "x");

Template (upQuot, "/");

Template (int_str, "lit");

Explanation ("incomplete program", ["error", 0]);

Explanation ("division by zero", ["error", 1]).

!     the Rules

define{root, "PI2Rules",

<<

! Evaluator Rules

! Constant nodes

if *Eval(e,c), Con(e), Litval(v,e), Meaning(f, "lit")

-> Value(f[v], e, c);

! Appl nodes

if *Eval(e,c), Appl(e), Left(x,e), Right(y,e)

-> Eval(x,c), Eval(y,c);
```

```
if *Value(u,x,c), *Value(v,y,c),
   Appl(e), Op(n,e), Left(x,e), Right(y,e), Meaning(f, n)
-> Check(f[u,v], e, c);

! Error Checking

if *Check(w, e, c), ~IsErrorcode[w]
-> Value(w, e, c);

if *Check(w, e, c), IsErrorcode[w], Explanation(s, w), *CurrentNode(q)
-> displayn{s}, CurrentNode(e), CurrentContext(c);

! Unparser

! Constant Nodes

if *Unparse(e), Con(e), Litval(v,e), Template(f, "lit")
-> Image(f[v], e);

! Identifier nodes

! Appl nodes

if *Unparse(e), Appl(e), Left(x,e), Right(y,e)
-> Unparse(x), Unparse(y);

if *Image(u,x), *Image(v,y),
   Appl(e), Op(n,e), Left(x,e), Right(y,e), Template(f, n)
-> Image(f[u,v], e);

! Command Interpreter Rules

! evaluate Command

if *Command("evaluate"), CurrentNode(E), CurrentContext(C)
```

```
-> Eval(E,C), EvalPending(E);

if *Value(V,E,C), *EvalPending(E)
-> displayn {V};

! return Command

if *Command("val"), *Argument(V), CurrentNode(E)
-> Value(V,E,C);

! show Command

if *Command("show"), CurrentNode(E)
-> Unparse(E), ShowPending(E);

if *Image(S,E), *ShowPending(E)
-> displayn{S};

! abort Command

if Command("abort"), *Eval(E,C) -> ;

if Command("abort"), *Value(V,E,C) -> ;

if Command("abort"), *Check(V,E,C) -> ;

if Command("abort"), *Nonlocal(C,D) -> ;

if Command("abort"), *Binds(D,N,V) -> ;

if *Command("abort"),
 ~Eval(E,C), ~Value(V,E,C), ~Nonlocal(C,D), ~Binds(D,N,V)
-> displayn{"aborted"};

! Handle incomplete program
```

```
if *Eval(E,C), Undef(E), *CurrentNode(Q)
-> displayn("Incomplete"), CurrentNode(E), CurrentContext(C);

if *Unparse(E), Undef(E)
-> Image ("< expr> ", E);

! Syntax Directed Editing

! in Command

if *Command("in"), *CurrentNode(E), Left(X,E)
-> CurrentNode(X), Command("show");

if *Command("out"), *CurrentNode(X), Left(X,E)
-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(Y), Right(Y,E)
-> CurrentNode(E), Command("show");

! next Command

if *Command("next"), *CurrentNode(X), Left(X,E), Right(Y,E)
-> CurrentNode(Y), Command("show");

! prev Command

if *Command("prev"), *CurrentNode(Y), Right(Y,E), Left(X,E)
-> CurrentNode(X), Command("show");

! delete command

if *Command("delete"), CurrentNode(E), *Con(E), *Litval(V,E)
-> Undef(E), Command("show");

if *Command("delete"), CurrentNode(E),
```

```
  *Appl(E), *Op(N,E), *Left(X,E), Right(Y,E)
-> Undef(E), Command("show");

if *Command("delete"), CurrentNode(E), Undef(E)
-> displayn("already deleted");

! # Command

if *Command("#"), *Argument(V), IsInt[V], CurrentNode(E), *Undef(E)
-> Con(E), Litval(V,E), Command("show");

if *Command("#"), *Argument(V), CurrentNode(E), ~Undef(E)
-> displayn("defined node");


! +, -, x , / Commands

if *Command(op), member [op, ["+ ", "-", "x", "/"]],
   *CurrentNode(E), *Undef(E)
-> CreateAppl(op, E, newobj{}, newobj{});

if *CreateAppl(op,E,X,Y)
-> Appl(E), Op(op,E), Left(X,E), Right(Y,E),
  Undef(X), Undef(Y), CurrentNode(X);

! begin Command

if *Command("begin"), *CurrentNode(Q)
->  CreateRoot(newobj{});

if *CreateRoot(E)
-> Root(E), Undef(E), CurrentNode(E);

! root Command
```

```
if *Command("root"), *CurrentNode(Q), Root(E)
-> CurrentNode(E), Command("show");

! Unparsing Variables

if *Unparse(E), Var(E), Ident(N,E)
-> Image(N,E);

! Unparsing Blocks

! Analysis

if *Unparse(E), Block(E), BndVal(X,E), Body(B,E)
-> Unparse(X), Unparse(B);

! Synthesis

if Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),
 *Image(U,X), *Image(V,B)
-> Image( TabIn + NL
  + "[let "+ N + "= "+ U
  + TabIn + NL + V + "]"
  + TabOut + TabOut,
 E);

! Editor

! var Command

if *Command("var"), *Argument(N), CurrentNode(E), *Undef(E)
-> Var(E), Ident(N,E);

! let Command
```

```
if *Command("let"), *Argument(N), *CurrentNode(E), *Undef(E)
-> CreateLet(N, E, newobj{}, newobj{});

if *CreateLet (N, E, X, B)
-> Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),
 Undef(X), Undef(B), CurrentNode(X);

! in Command for Blocks

if *Command("in"), *CurrentNode(E), BndVal(X,E)
-> CurrentNode(X), Command("show");

! out Command for Blocks

if *Command("out"), *CurrentNode(X), BndVal(X,E)
-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(B), Body(B,E)
-> CurrentNode(E), Command("show");

! next Command for Blocks

if *Command("next"), *CurrentNode(X),
 BndVal(X,E), Body(B,E)
-> CurrentNode(B), Command("show");

! prev Command for Blocks

if *Command("prev"), *CurrentNode(B),
 Body(B,E), BndVal(X,E)
-> CurrentNode(X), Command("show");

! EVALUATION OF BLOCK
```

```
! Evaluate Bound Value

if *Eval(E,C), Block(E), BndVal(X,E)
-> Eval(X,C);

! Bind Var. to Value
! Evaluate Body

if Block(E), BndVar(N,E), BndVal(X,E), Body(B,E),
 *Value(V,X,C)
-> CreateContext (newobj{}, N, V, C, B);

if *CreateContext(D,N,V,C,B)
-> Context(D), Binds(D,N,V), Nonlocal(C,D), Eval(B,D);

! Return Value To Surrounding Block

if Block(E), Body(B,E),
 *Value(V,B,D), *Nonlocal(C,D), *Binds(D,N,W), *Context(D)
-> Value(V,E,C);

! Variable Lookup

if *Eval(E,C), Var(E), Ident(N,E)
-> Looking(N,C,E,C);

! VARIABLE EVALUATION

! Binding Found

if *Looking(N,C,E,D), Binds(C,N,V)
-> Value(V,E,D)

! Continue Looking
```

```
else if *Looking(N,C,E,D), Nonlocal(Cprime,C)

-> Looking(N,Cprime,E,D)

! Variable Unbound

else if *Looking(N,C,E,D), *CurrentNode(Q), *CurrentContext(Q)

-> CurrentNode(E), CurrentContext(D), displayn("Unbound: " + N);

! Debugging Commands

! context Command

if *Command("context"), CurrentContext(C), Binds(C,N,V)

-> displayn( N + " = " + int_str[V] )

else if *Command("context")

-> displayn("no bindings");

! out_context Command

if *Command("out_context"), *CurrentContext(D), Nonlocal(C,D)

-> CurrentContext(C), Command("context")

else if *Command("out_context")

-> displayn ("at outermost level");

! in_context Command

if *Command("in_context"), *CurrentContext(C), Nonlocal(C,D)

-> CurrentContext(D), Command("context")

else if *Command("in_context")

-> displayn("at innermost level");

! alter Command
```

```
if *Command("alter"), *Argument(U),
 CurrentContext(C), *Binds(C,N,V)
-> Binds(C,N,U), Command("context")

else if *Command("alter"), *Argument(Q)
-> displayn("no binding");

! Test Driver

if *Script(A,Nil) -> A("Script completed")

else if *Script(A,L), member [first[L], ["#", "val", "let", "var", "alter"]]
-> { display {"... "};
     display {first [rest [L]]};
     displayn {" " + first [L]};
     Command(first[L]), Argument(first[rest[L]]);
     PendScript(A, rest[rest[L]]) }

else if *Script(A,L)
-> { displayn {"... " + first [L]};
     Command(first[L]);
     PendScript(A, rest[L]) };

if *PendScript(A,L), ~Command(Q) -> Script(A,L);

if *Test(A,1) -> {
  Script{["begin","let","X","+ ","#",3,"next","#",5,"out","next",
    "let","Y","/","#",6,"next","#",2,"out","next",
    "+ ","var","X","next","/","var","Y","next","#",1,"root","evaluate"]};
  A("Test done");
  };
```

```
if *Test(A,2) -> {
  Script{["in","next","in","next","in","next","in","next",
    "delete","#",0,"root","evaluate"]};
  A("Test done");
  };

if *Test(A,3) -> {
  Script{["context","out_context","alter",7,
    "in_context","out_context","out_context"]};
  A("Test done");
  }

>> }.

!       activate the rules

act{ PI2Rules }.
CurrentNode(Nil).
CurrentContext(Nil).
displayn{"PI-2 System loaded"}.
```

APPENDIX B: Transcript of Ω Session

The following is a transcript of an Ω session illustrating the operation of the prototype programming environment shown in Appendix A. The call 'Test {*n*}' causes the commands comprising the *n*th test script to be executed in order. Each command is printed on a separate line, followed by whatever output is generated by the programming environment. This transcript was produced by the McArthur interpreter [McArthur84].

```
% om
OMEGA-1   11/30/84

Use Cntl-D or exit{} to quit.

For help, enter help{"?"}.

To report a bug, enter Bugs{}.
> do{"PI2.rul"}.
PI-2 System loaded
OK
> Test{1}.
 ... begin
 ... X let
 ... +
 ... 3 #
3
 ... next
< expr>
 ... 5 #
5
 ... out
(3 + 5)
```

```
... next
< expr>
... Y let
... /
... 6 #
6
... next
< expr>
... 2 #
2
... out
(6 / 2)
... next
< expr>
... +
... X var
... next
< expr>
... /
... Y var
... next
< expr>
... 1 #
1
... root
[let X = (3 + 5)
  [let Y = (6 / 2)
    (X + (Y / 1)) ] ]
```

```
... evaluate
11
Test done
> Test{2}.
... in
(3 + 5)
... next
[let Y = (6 / 2)
 (X + (Y / 1)) ]
... in
(6 / 2)
... next
(X + (Y / 1))
... in
X
... next
(Y / 1)
... in
Y
... next
1
... delete
< expr>
... 0 #
0
... root
[let X = (3 + 5)
 [let Y = (6 / 2)
```

```
   (X + (Y / 0)) ] ]
... evaluate
division by zero
Test done
> Test{3}.
... context
Y = 3
... out_context
X = 8
... 7 alter
X = 7
... in_context
Y = 3
... out_context
X = 7
... out_context
no bindings
Test done
> exit{}.

Goodbye.
%
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 — 2

Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943 — 2

Office of Research Administration
Code 012A
Naval Postgraduate School
Monterey, CA 93943 — 1

Chairman, Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 — 40

Associate Professor Bruce J. MacLennan
Code 52ML
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943 — 12

Dr. Robert Grafton
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217-5000 — 1

Dr. David Mizell
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106 — 1

Dr. Stephen Squires
DARPA
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209 — 1

Professor Jack M. Wozencraft, 62Wz
Department of Electrical and Comp. Engr.
Naval Postgraduate School
Monterey, CA 93943 — 1

Professor Rudolf Bayer
Institüt für Informatik
Technische Universität
Postfach 202420
D-8000 Munchen 2
West Germany — 1

Dr. Robert M. Balzer
USC Information Sciences Inst.
4676 Admiralty Way
Suite 10001
Marina del Rey, CA 90291 1

Mr. Ronald E. Joy
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402 1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain 1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152 1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040 1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714 1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011 1

Professor S. Ceri
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy 1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720 1

Antonio Corradi
Dipartimento di Elettronica
Informatica e Sistemistica
Universita Degli Studi di Bologna
Viale Risorgimento, 2

Bologna
Italy 1

Dr. Peter J. Welcher
Mathematics Dept., Stop 9E
U.S. Naval Academy
Annapolis, MD 21402 1

Dr. John Goodenough
Wang Institute
Tyng Road
Tyngsboro, MA 01879 1

Professor Richard N. Taylor
Computer Science Department
University of California at Irvine
Irvine, CA 92717 1

Dr. Mayer Schwartz
Computer Research Laboratory
MS 50-662
Tektronix, Inc.
Post Office Box 500
Beaverton, OR 97077 1

Professor Lori A. Clarke
Computer and Information Sciences Department
LGRES ROOM A305
University of Massachusetts
Amherst, MA 01003 1

Professor Peter Henderson
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794 1

Dr. Mark Moriconi
SRI International
333 Ravenswood Avenue
Manlo Park, CA 95025 1

Professor William Waite
Department of Electrical and Computer Engineering
The University of Colorado
Campus Box 425
Boulder, CO 80309-0425 1

Professor Mary Shaw
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213 1

Dr. Warren Teitelman
Engineering/Software
Sun Microsystems Federal, Inc.
2550 Garcia Avenue
Mountain View, CA 94031